EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Homework 10

This homework is due on Friday, April 21, 2023, at 10:59PM.

1. Meta-learning for Learning 1D functions

A common toy example with Neural Networks is to learn a 1D function. Suppose now that our task is not to learn not just one 1D function, but any of a class of 1D functions drawn from a *task distribution* $\mathcal{D}_T$.

In this problem we consider all signals of the form

$$y = \sum_{s \in \mathcal{S}} \alpha_s \phi_s^u(x)$$

The task distribution produces individual tasks which have true features with random coefficients in some *a priori* unknown set of indices $\mathcal{S}$. We do not yet know the contents of $\mathcal{S}$, but we can sample tasks from $\mathcal{D}_T$.

The important question is thus, **how do we use sampled tasks in training to improve our performance on an unseen task drawn from $\mathcal{D}_T$ at test time?**

One solution is to use our training tasks to learn a set of weights to apply to the features before performing regression through meta-learning. That is, we choose feature weights c_k to apply to the features $\phi_k^u(x)$ before learning coefficients $\hat{\beta}_k$ such that

$$\hat{y} = \sum_{k=0}^{d-1} \hat{\beta}_k c_k \phi_k^u(x).$$

These feature weights c_k are a toy model for the deep network that precedes the task-specific final layer in meta-learning.

We can then perform the min-norm optimization

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{\beta}\|_2^2 \tag{1}$$

$$\text{s.t. } \mathbf{y} = \sum_{k=0}^{d-1} \beta_k c_k \boldsymbol{\Phi}_k^u \tag{2}$$

where $\boldsymbol{\Phi}^u$ is the column vector of features $[\phi_0^u(x), \phi_1^u(x), \ldots, \phi_{d-1}^u(x)]^\top$ which are orthonormal with respect to the test distribution.

Now, we want to **learn c** which minimizes the expectation for $\hat{\boldsymbol{\beta}}$ over all tasks,

$$\underset{\mathbf{c}}{\operatorname{argmin}} \mathbb{E}_{\mathcal{D}_T} \left[\mathcal{L}_T \left(\hat{\boldsymbol{\beta}}_T, \mathbf{c} \right) \right]$$

where $\mathcal{L}_T \left(\hat{\boldsymbol{\beta}}_T, \mathbf{c} \right)$ is the loss from learning $\hat{\boldsymbol{\beta}}$ for a specific task with the original formulation and a given $\mathbf{c}$ vector. $\mathbf{c}$ is shared across all tasks and is what we will optimize with meta-learning.

There are many machine learning techniques which can fall under the nebulous heading of meta-learning, but we will focus on one with Berkeley roots called Model Agnostic Meta-Learning (MAML)[1] which optimizes the initial weights of a network to rapidly converge to low loss within the task distribution. The MAML algorithm as described by the original paper is shown in Fig. 1.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figure 1: MAML algorithm. We will refer to the training steps on line 6 as the *inner update*, and the training step on line 8 as the *meta update*.

At a high level, MAML works by sampling a "mini-batch" of tasks $\{T_i\}$ and using regular gradient descent updates to find a new set of parameters θ_i for each task starting from the same initialization θ. Then the gradient w.r.t. the original θ each calculated for each task using the task-specific updated weights θ_i, and θ is updated with these 'meta' gradients. Fig. 2 illustrates the path the weights take with these updates.

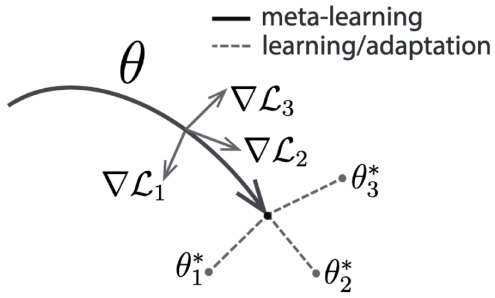

Figure 2: MAML gradient trajectory illustration

The end goal is to produce weights θ^* which can reach a state useful for a particular task from $\mathcal{D}_T$ after a few steps — needing to use less data to learn. If you want to understand the fine details of the algorithm and implementation, we recommend reading the original paper and diving into the code provided with this problem.

(a) In the original MAML algorithm, the inner loop performs gradient descent to optimize loss with respect to a task distribution. However, here we're going to use the closed form min-norm solution for regression instead of gradient descent.

[1] C. Finn, P. Abbeel, S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017*

Let's recall the closed form solution to the min-norm problem. Write the solution to

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{\beta}\| \text{ , such that } \mathbf{y} = A\boldsymbol{\beta}$$

in terms of A and $\mathbf{y}$.

Solution:

$$\hat{\boldsymbol{\beta}} = A^\top (AA^\top)^{-1}\mathbf{y}$$

is the min-norm solution.

(b) For simplicity, suppose that we have exactly one training point (x, y), and one true feature $\phi_t^u(x) = \phi_1^u(x)$. We have a second (alias) feature that is identical to the first true feature, $\phi_a^u(x) = \phi_2^u(x) = \phi_1^u(x)$. This is a caricature of what always happens when we have fewer training points than model parameters.

The function we wish to learn is $y = \phi_t^u(x)$. We learn coefficients $\hat{\boldsymbol{\beta}}$ using the training data. Note, both the coefficients and the feature weights are 2-d vectors.

Show that in this case, the solution to the min-norm problem 1 is $\hat{\boldsymbol{\beta}} = \frac{1}{c_0^2+c_1^2}\begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$

Solution: Since $\phi_1^u(x) = \phi_2^u(x) \equiv \phi^u(x)$, we plug into the min-norm solution which is

$$\begin{aligned}
\hat{\boldsymbol{\beta}} &= \begin{bmatrix} c_0\phi^u(x) \\ c_1\phi^u(x) \end{bmatrix} \left(\begin{bmatrix} c_0\phi^u(x) & c_1\phi^u(x) \end{bmatrix} \begin{bmatrix} c_0\phi^u(x) \\ c_1\phi^u(x) \end{bmatrix} \right)^{-1} y \\
&= \phi^u(x)^2 \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \left(\phi^u(x)^2 \left(c_0^2 + c_1^2 \right) \right)^{-1} \\
&= \frac{1}{c_0^2 + c_1^2} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}
\end{aligned}$$

The least-squares $\hat{\boldsymbol{\beta}}$ subject to the constraint $\phi^u(x) = \beta_0 c_0 \phi^u(x) + \beta_1 c_1 \phi^u(x)$ (which you could also find by using the Lagrangian and KKT conditions) is

$$\hat{\beta}_0 = \frac{c_0}{c_0^2 + c_1^2}$$

$$\hat{\beta}_1 = \frac{c_1}{c_0^2 + c_1^2}$$

(c) Assume for simplicity that we have access to infinite data from the test distribution for the purpose of updating the feature weights $\mathbf{c}$. **Calculate the gradient of the expected test error with respect to the feature weights c_0 and c_1, respectively:**

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{c}} \left(\mathbb{E}_{x_{test}, y_{test}} \left[\frac{1}{2} \left\| y - \hat{\beta}_0 c_0 \phi_t^u(x) - \hat{\beta}_1 c_1 \phi_a^u(x) \right\|_2^2 \right] \right).$$

Use the values for $\boldsymbol{\beta}$ from the previous part. *(Hint: the features $\phi_i^u(x)$ are orthonormal under the test distribution. They are not identical here.)*

Solution: Plugging $\hat{\boldsymbol{\beta}}$ from part (b) into the expected test error, we have

$$\mathbb{E}_{x_{test},y_{test}}\left[\frac{1}{2}\left\|y-\frac{c_0^2}{c_0^2+c_1^2}\phi_t^u(x)-\frac{c_1^2}{c_0^2+c_1^2}\phi_a^u(x)\right\|_2^2\right]$$

Now we use the orthogonality of the features under the test distribution to evaluate the expected norm.

$$\begin{aligned}\mathcal{E} &= \mathbb{E}_{x_{test},y_{test}}\left[\frac{1}{2}\left\|\phi_t^u(x)-\frac{c_0^2}{c_0^2+c_1^2}\phi_t^u(x)-\frac{c_1^2}{c_0^2+c_1^2}\phi_a^u(x)\right\|_2^2\right]\\ &= \frac{1}{2}\left(1-\frac{c_0^2}{c_0^2+c_1^2}\right)^2+\frac{1}{2}\left(\frac{c_1^2}{c_0^2+c_1^2}\right)^2\\ &= \left(\frac{c_1^2}{c_0^2+c_1^2}\right)^2\\ &= \frac{c_1^4}{(c_0^2+c_1^2)^2}\end{aligned}$$

Calculating the gradient, we have

$$\begin{aligned}\frac{\mathrm{d}\mathcal{E}}{\mathrm{d}c_0} &= \frac{-4c_0c_1^4}{\left(c_0^2+c_1^2\right)^3}\\ \frac{\mathrm{d}\mathcal{E}}{\mathrm{d}c_1} &= \frac{4c_0^2c_1^3}{\left(c_0^2+c_1^2\right)^3}\end{aligned}$$

(d) **Generate a plot** showing that, for some initialization $\mathbf{c}^{(0)}$, as the number of iterations $i \to \infty$ the weights empirically converge to $c_0 = \|\mathbf{c}^{(0)}\|$, $c_1 = 0$ using gradient descent with a sufficiently small step size. Include the initialization and its norm and the final weights. What will $\boldsymbol{\beta}$ go to?

Solution:

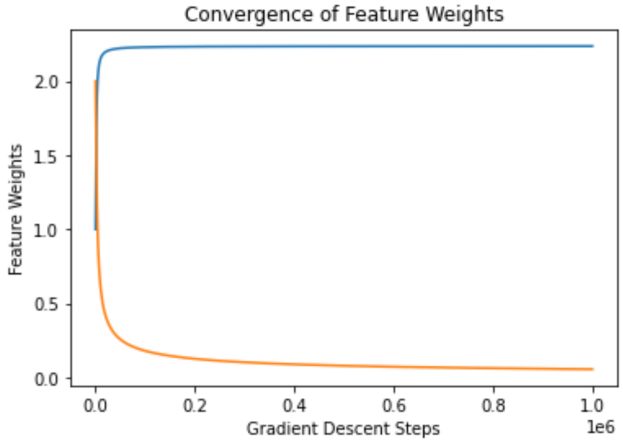

Figure 3: Asymptotic behavior of the feature weights with learning rate 0.001. $c_1 \to \|\mathbf{c}^{(0)}\|$ and $c_1 \to 0$.

As $c_1 \to 0$, $\hat{\beta}_0 \to \frac{1}{c_0}$ and $\hat{\beta}_1 \to 0$.

Run the code in the Jupyter Notebook and then answer these questions:

(e) (In MAML for regression using closed-form solutions) Considering the plot of regression test loss versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are set to 1?** Additionally, **how does their performance compare to the oracle**, which performs regression using only the features present in the data? Can you **explain the reason for the downward spike observed at `n_train_post = 32`**?

Solution: By looking at the plot of test loss vs n_train_post we see that using the meta-learned feature weights the test performance is better than when we use the all 1s feature weights but worse than the oracle. As we increase n the effect of feature weights is less prominent and in all cases our test error goes down. There is a prominent downward spike at n_train_post = 32 since we do the training on inner tasks with n_train_inner = 32. The feature weights evolve during the meta learning process to downweight the aliases of the true feature towards 0 (when n = 32) which results in the performance being close to that of the oracle for this particular value of n.

(f) By examining the changes in feature weights over time during meta-learning, **can you justify the observed improvement in performance?** Specifically, can you **explain why certain feature weights are driven towards zero**?

Solution: We see that the weight on the favored features grows throughout the meta learning process but what is unique to this setting is the weights on the aliases of the true feature (corresponding to n=32) are getting downweighted towards 0. The weights on the other features are largely unchanged.

(g) (In MAML for regression using gradient descent) **With `num_gd_steps = 5`, does meta-learning contribute to improved performance during test time? Furthermore, if we change `num_gd_steps` to 1, does meta-learning continue to function effectively?**

Solution: We observe that for both values of num_gd_steps (1 and 5) meta learning helps us improve performance. From the previous cell observe that with num_gd_steps=1, the solution using gradient descent is not the same as the closed form solution but this does not deter the meta learning procedure. For a sanity check you can try running the cell with num_gd_steps = 0 and observe that in this case we don't learn anything.

(h) (In MAML for classification) Based on the plot of classification error versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are 1? How does the performance of the meta-learned feature weights compare to the oracle** (which performs logistic regression using only the features present in the data)?

Solution: By looking at the plot of classification error vs n_train_post we see that using the meta-learned feature weights the test performance is better than when we use the all 1s feature weights but worse than the oracle. As we increase n the effect of feature weights is less prominent and in all cases our classification error goes down.

(i) By observing the evolution of the feature weights over time as we perform meta-learning, **can you justify the improvement in performance? Specifically, can you explain why some feature weights are being driven towards zero?**

Solution: We see that the weight on the favored features grows throughout the meta learning process and the weights on the other features are slightly decreased. The feature weight vector after meta learning is moving towards the oracle feature weights (1 on favored features and 0 on other featuers) and this explains the improvement in performance.

2. Vision Transformer

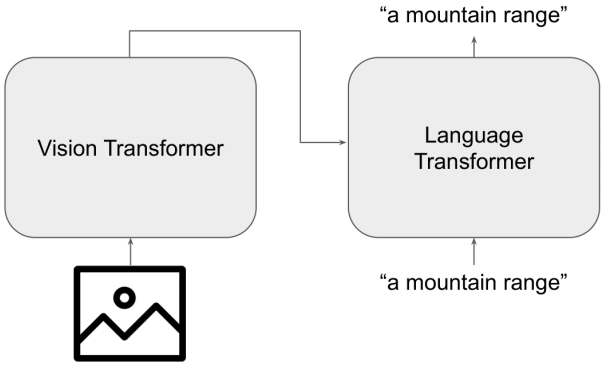

Figure 4: Image captioning model

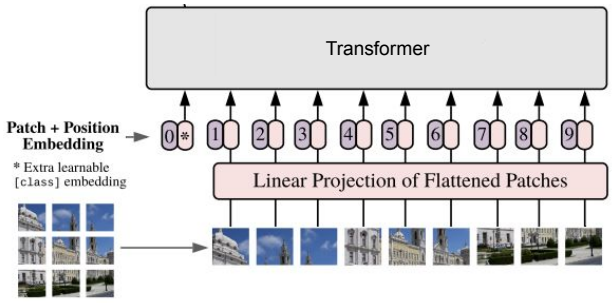

Figure 5: Vision Transformer

You are building a model to perform image captioning. As shown in Figure 4, the model consists of a vision transformer which takes in images and a language transformer which outputs captions. The language transformer will use cross-attention to access the representation of the image.

(a) For each transformer, state whether it is more appropriate to use a transformer encoder (a transformer with no masking except to handle padding) or decoder (a transformer with autoregressive self-attention masking) and why.

Vision transformer?

◯ Encoder-style transformer
◯ Decoder-style transformer

Reason:

Language transformer?

◯ Encoder-style transformer
◯ Decoder-style transformer

Reason:

Solution: You should use an encoder for the vision transformer since we are just trying to produce an image representation. You should use a decoder for the language transformer since you need to generate tokens autoregressively.

(b) A standard language transformer for captioning problems alternates between layers with cross-attention between visual and language features and layers with self-attention among language features. Let's say we modify the language transformer to have a single layer which performs both attention operations at once. The grid below shows the attention mask for this operation. (For now, assume the vision transformer only outputs 3 image tokens called <ENC1>, <ENC2>, and <ENC3>. <SOS> is the start token, and <PAD> is a padding token.)

(i) One axis on this grid represents sequence embeddings used to make the queries, and the other axis represents sequence embeddings used to make the keys. **Which is which?**

◯ Each column creates a query, each row creates a key and a value

Solution: Correct, we need to have keys and values corresponding to the image encodings so that they are queryable by the cross-attention mechanism in the decoder process generating the caption. The queries must be for the columns since those correspond to the language model emitting a caption.

◯ Each column creates a key and a value, each row creates a query

◯ Each column creates a query and a value, each row creates a key

◯ Each column creates a key, each row creates a query and a value

(ii) **Mark X in some of the blank cells in the grid to illustrate the attention masks.** (A X marked cell is masked out, a blank cell is not.)

	<SOS>	a	mountain	range	<PAD>
<SOS>					
a					
mountain					
range					
<PAD>					
<ENC1>					
<ENC2>					
<ENC3>					

Solution:

	<SOS>	a	mountain	range	<PAD>
<SOS>					X
a	X				X
mountain	X	X			X
range	X	X	X		X
<PAD>	X	X	X	X	X
<ENC1>					X
<ENC2>					X
<ENC3>					X

Note: the padding is always masked out since there is no point in trying to see what is there. The rest of the masking reflects the autoregressive (causal) nature of caption generation — we can't access the representations of tokens that won't yet be generated at test time.

(c) In discussion, we showed that the runtime complexity of vision transformer attention is $O(D(H^4/P^4))$, where H is the image height and width, P is the patch size, and D is the feature dimension of the queries, keys, and values. Some recent papers have reduced the complexity of vision transformer attention by segmenting an image into windows, as shown in Figure 6.

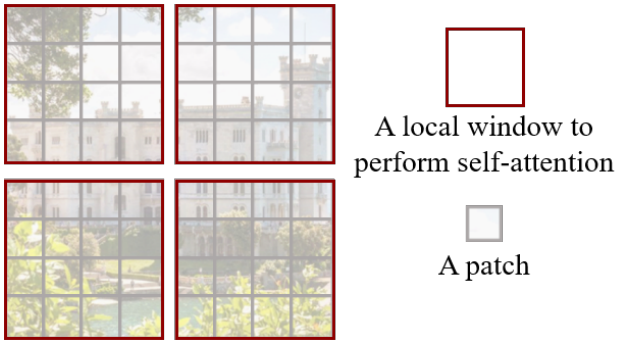

Figure 6: Vision transformer attention with windows

Patches only attend to other patches within the same window. **What is the Big-O runtime complexity of the attention operation after this modification?** Assume each window consists of K by K patches.

Solution: There are $(H/P)^2$ items in the sequence. and each will attend to K^2 other patches. All vectors are size D. Combining this, we get $O(\frac{H^2}{P^2}K^2D)$.

3. Pretraining and Finetuning

When we use a pretrained model without fine-tuning, we typically just train a new task-specific head. With standard fine-tuning, we also allow the model weights to be adapted.

However, it has recently been found that we can selectively fine-tune a subset of layers to get better performance especially under certain kinds of distribution shifts on the inputs. Suppose that we have a ResNet-26 model pretrained with CIFAR-10. Our target task is CIFAR-10-C, which adds pixel-level corruptions (like adding noise, different kinds of blurring, pixelation, changing brightness and contrast, etc) to CIFAR-10. If we could only afford to fine-tune one layer, **which layer (i.e. 1,2,3,4,5) in Figure 7 should we choose to finetune to get the best performance on CIFAR-10-C? Give brief intuition as to why.**

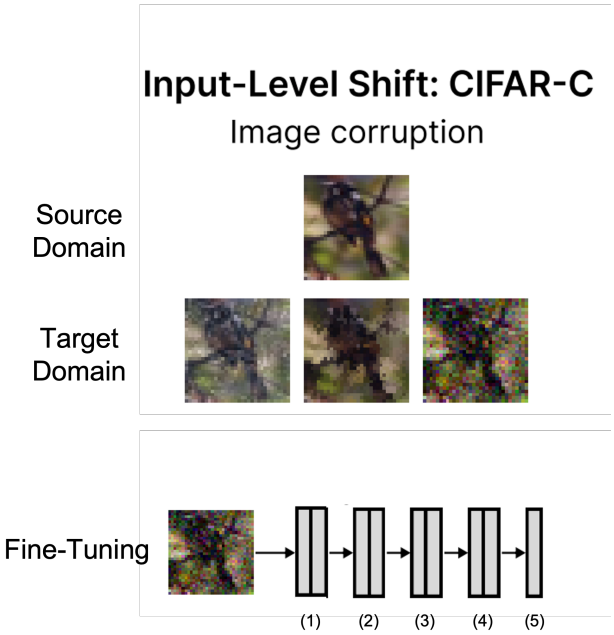

Figure 7: Fine-tuning the model pretrained with CIFAR-10 on CIFAR-10-C dataset

Solution: (1). Early layers of convolutional neural network extract low-level features of image. CIFAR-10-C is low level feature shifts. Therefore, fine-tuning only the first block of the pretrained model outperforms the traditional approach of adjusting the task-head. https://arxiv.org/pdf/2210.11466.pdf

4. Prompting Language Models

(a) **Exploring Pretrained LMs**

Play around with the web interface at https://dashboard.cohere.ai/playground/generate. This plaground provides you an interface to interact with a large language model from Cohere and tweak various parameters. You will need to sign up for a free account.

Once you're logged in, you can choose a model in the parameters pane on the right. "command-xlarge-nightly" is a generative model that responds well with instruction-like prompts. "xlarge" and "medium" are generative models focusing on sentence completion. Spend a while exploring prompting these models for different tasks. Here are some suggestions:

- Look through the 'Examples . . . ' button at the top of the page for example prompts.
- Ask the model to answer factual questions.
- Prompt the model to generate a list of 100 numbers sampled uniformly between 0 and 9. Are the numbers actually randomly distributed?
- Insert a poorly written sentence, and have the model correct the errors.
- Have the model brainstorm creative ideas (names for a storybook character, recipes, solutions to solve a problem, etc.)
- Chat with the model like a chatbot.

Answer the questions below:

i. **Describe one new thing you learned by playing with these models.**

Solution: Answers may vary.

ii. **How does the temperature parameter affect the outputs?** Justify your answer with a few examples.

Solution: When temperature = 0, the model is deterministic and outputs the greedy argmax answer every time you generate with the same prompt. When the temperature is higher, results are different every time, and the model is more likely to have weird, creative, and nonsensical outputs.

iii. **Describe a task where the larger models (e.g., "xlarge" or "command-xlarge-nightly") significantly outperform the smaller ones (e.g., "medium").** Paste in examples from the biggest and smallest model to show this.

Solution: Answers may vary.

iv. **Describe a task where even the largest model performs badly**. Paste in an example to show this.

Solution: Answers may vary

v. **Describe a task where the model's outputs improve significantly with few-shot prompting compared to zero-shot prompting.**

Solution: Answers may vary, but this is often the case when you want anwers to be in a specific output format.

(b) **Using LMs for classification**

Run `lm_prompting.ipynb`, then answer the following questions. If you did not do part (a), you will still need to get a Cohere account to complete this part.

i. Analyze the command-xlarge-nightly model's failures. **What kinds of failures do you see with different prompting strategies?**

Solution: For SimplePrompt and SimpleQA prompt, many of the errors are the model outputting invalid solutions (especially newline characters with the SimplePrompt.)
For QAInstruction and the two FewShot variants, failures are mostly choosing the incorrect answer. A large portion of the incorrect answers are choices which seem reasonable even to a human.

ii. **Does providing correct labels in few-shot prompting have a significant impact on accuracy?**

Solution: Answers may vary depending on how many data points you use, but you should see that the accuracy with incorrect labels in the prompt is similar to or slightly worse than with clean prompts. (Confidence decreases slightly too.)

iii. **Observe the model's log probabilities. Does it seem more confident when it is correct than when it is incorrect?**

Solution: The model is on average more confident when it is correct, though which prompt strategy is being used is more correlated with confidence than correctness/incorrectness.

iv. **Why do you think the GPT2 model performed so much worse than the command-xlarge-nightly model on the question answering task?**

Solution: The GPT2 model is much smaller and trained on less data.

v. **How did soft prompting compare to hard prompting on the pluralize task?**

Solution: At convergence, the soft prompt significantly outperforms hard prompts, even with several examples in the hard prompt.

vi. You should see that when the model fails (especially early in training of a soft prompt or with a bad hard prompt) it often outputs common but uninformative tokens such as the, ", or \n. **Why does this occur?**

Solution: When the model is uncertain which token comes next, the most likely token is often a token which commonly occurs throughout most text corpuses.

5. Soft-Prompting Language Models

You are using a pretrained language model with prompting to answer math word problems. You are using chain-of-thought reasoning, a technique that induces the model to "show its work" before outputting a final answer.

Here is an example of how this works:

```
[prompt] Question: If you split a dozen apples evenly among yourself
and three friends, how many apples do you get? Answer: There are 12
apples, and the number of people is 3 + 1 = 4. Therefore, 12 / 4 = 3.
Final answer: 3\n
```

If we were doing hard prompting with a frozen language model, we would use a hand-designed [prompt] that is a set of tokens prepended to each question (for instance, the prompt might contain instructions for the task). At test time, you would pass the model the sequence and end after "Answer:" The language model will continue the sequence. You extract answers from the output sequence by parsing any tokens between the phrase "Final answer: " and the newline character "\n".

(a) Let's say you want to improve a frozen GPT model's performance on this task through soft prompting and training the soft prompt using a gradient-based method. This soft prompt consists of 5 vectors prepended to the sequence at the input — these bypass the standard layer of embedding tokens into vectors. (Note: we do not apply a soft prompt at other layers.) Imagine an input training sequence which looks like this:

```
["Tokens" 1-5: soft prompt] [Tokens 6-50: question]
[Tokens 51-70: chain of thought reasoning]
[Token 71: answer] [Token 72: newline]
[Tokens 73-100: padding].
```

We compute the loss by passing this sequence through a transformer model and computing the cross-entropy loss on the output predictions. If we want to train the soft-prompt to output correct reasoning and produce the correct answer, **which output tokens will be used to compute the loss?** (Remember that the target sequence is shifted over by 1 compared to the input sequence. So, for example, the answer token is position 71 in the input and position 70 in the target).

Solution: We include output tokens 50-71 (the chain of thought, answer, and newline) in the loss.

In more depth:

- Output tokens 1-4 - soft prompt: there is no ground truth output for these tokens, so we cannot train a loss with them.
- Output tokens 5-49 - question: we technically have ground-truth here we could use for the loss, but there is no point in training the model to be good at generating questions since at test-time it will be given the question and only needs to output the reasoning and answer.
- Output tokens 50-69 - reasoning: This is important to include if we want the model to learn to output reasoning. (There may be multiple forms of valid reasoning which could go here, but we still can train on the reasoning examples in our training set.)

- Output token 70 - answer: This is important to include in the loss.
- Output token 71 - newline: We include this in the loss too. If the model is not trained to output a newline after it finishes producing the answer, then when we parse the answers we will get extra tokens at the end.
- Output tokens 72-100: This is padding, so we don't need to include this.

(Note that the indices provided were for **this particular example**. Other training examples in the dataset will have different length questions, reasoning, and answers.)

(b) Continuing the setup above, **how many parameters are being trained in this model?** You may write this in terms of the max sequence length S, the token embedding dimension E, the vocab size V, the hidden state size H, the number of layers L, and the attention query/key feature dimension D.

Solution: The gradient updates will be applied to the 5 vectors that constitute the soft prompt. Everything else in the model will be frozen.

There are 5 vectors of size E, so 5E total.

(c) **Mark each of the following statements as True or False and give a brief explanation.**

(i) If you are using an autoregressive GPT model as described in part (a), it's possible to precompute the representations at each layer for the indices corresponding to prompt tokens (i.e. compute them once for use in all different training points within a batch).

Solution: True, since the masking is autoregressive, the prompt representations will be the same for each data point.

Note that during training, the fact that the inputs and representations are the same also means that the linear mapping (induced by the chain rule) that lets gradients get pulled back onto the parameters is also locked into place. Of course, the actual updates to parameters from those gradients have to be computed for each training point in the batch so that they can be combined to do an update, but this gradient update is not a representation at any layer.

(ii) If you compare the validation-set performance of the *best possible* K-token hard prompt to the *best possible* K-vector soft prompt, the soft-prompt performance will always be equal or better.

Solution: True, since the embedding of the best possible hard prompt is contained within the set of soft prompts, the best soft prompt must be at least as good.

(iii) If you are not constrained by computational cost, then fully finetuning the language model is always guaranteed to be a better choice than soft prompt tuning.

Solution: False, full finetuning, especially on a small dataset may hurt generalization and encourage overfitting. In practice, this tends not to happen often because of the regularizing effects of gradient-descent training in the context of so much overparameterization. What typically happens with full fine-tuning is that only a low-rank update actually ends up happening.

(iv) If you use a dataset of samples from Task A to do prompt tuning to generate a soft prompt which is only prepended to inputs of Task A, then performance on some other Task B with its own soft prompt might decrease due to catastrophic forgetting.

Solution: False, Task B performance is unaffected since the core model's parameters don't change. Those are frozen.

(d) Suppose that you had a family of related tasks for which you want use a frozen GPT-style language model together with learned soft-prompting to give solutions for the task. Suppose that you have substantial training data for many examples of tasks from this family. **Describe how you would adapt a meta-learning approach like MAML for this situation?**

(HINT: This is a relatively open-ended question, but you need to think about what it is that you want to

learn during meta-learning, how you will learn it, and how you will use what you have learned when faced with a previously unseen task from this family.)

Solution: The spirit of MAML is to train with an eye towards how you will act at test time. At test time, when faced with a previously unseen task from this family, we will initialize our soft-prompt to the known initialization. Then, we will use the training data to fine-tune the soft prompt to improve model performance by gradient descent. We'll see how we do on some held-out set.

Consequently, the thing we want to learn during meta-learning is the initial condition for the prompt.

How we could do it is to first take our training data for each task and split it into a train-train set and a held-out set. Then, we start with a random initialization for the parameters (we will note some variations later) defining the soft prompt start. Now, we begin to sample tasks where we:

- Initialize the soft prompt to our current soft-prompt-start and then;
- take a few steps of SGD using the task-specific loss and a random sampling of the training data.
- Having done that, we now use the held-out data to compute a gradient for the loss with respect to the intialization.
- We update the intialization of the soft-prompt-start by taking a step along this gradient
- Repeat with a freshly sampled task.

This updates the soft prompt initialization to an initialization that will hopefully work better after a few steps of training.

There are many variations on this basic MAML approach that can be done.

- We could use a mini-batch of tasks instead of sampling a single task.
- We could also allow our tasks to spawned partially trained copies and persist across meta-training iterations. (Essentially, after a task is used for a step of meta-training we flip a coin to decide whether to keep it or kill it. If we keep it, it gets added back to our pool of tasks with its now updated initialization intact. When it gets sampled again, it keeps its state but the gradient calculated using that is used to update the global soft-prompt-start — this incorporates ideas from "first-order MAML" and the partial spirit of approaches like REPTILE. Because only the original task can spawn copies, and each of the copies has a Poisson death in the training process, the expected number of tasks in our training pool stays bounded.)
- We could also combine with global pre-training where we use a fraction of the tasks to train a warm initializaiton using standard pre-training approaches (no MAML style gradients over unrolled gradient updates).
- We could also try to combine with k-means (or EM) style thinking and keep a number of initializations for the soft prompt in play. Then, when it comes to applying the gradient updates, we could have them softmaxed by the performance of that particular initialization on the held-out set. (This basically trains up a small library of initial soft-prompts with the expectation that one of them will hopefully be in the right neighborhood to rapidly train up a good soft-prompt for a new task.)
- You could further try to combine with human-designed/conjectured "hard prompts" for each task with the idea that upon encountering a new task, a human could be asked to guess a hard prompt for it. Then, the initialization of the soft-prompt before tuning could be a convex combination of the human hard-prompt with the MAML-learned good initialization. This can also be combined with a regularizer that prevents the finally learned soft-prompt from wandering too far away from the embedding of the human-provided hard prompt for the task.

The advantage of learning deep learning in the context of a solid grasp of machine learning fundamentals is that you as students should be able to generate many such variations yourselves. Not all of them

are going to work well and empirical testing is always required to make choices, but before you can try stuff our empirically you have to be able to come up with things to try.

6. TinyML - Quantization and Pruning.

(This question has been adapted with permission from MIT 6.S965 Fall 2022)

TinyML aims at addressing the need for efficient, low-latency, and localized machine learning solutions in the age of IoT and edge computing. It enables real-time decision-making and analytics on the device itself, ensuring faster response times, lower energy consumption, and improved data privacy.

To achieve these efficiency gains, techniques like quantization and pruning become critical. Quantization reduces the size of the model and the memory footprint by representing weights and activations with fewer bits, while pruning eliminates unimportant weights or neurons, further compressing the model.

(a) Please complete `pruning.ipynb`, then answer the following questions.

i. In part 1 the histogram of weights is plotted. **What are the common characteristics of the weight distribution in the different layers?**

Solution: The distribution of weights is centered on zero with tails dropping off quickly.

ii. **How do these characteristics help pruning?**

Solution: Weights that are close to 0 are the ones that are lease important, which helps alleviate the impact of pruining on model accuracy.

iii. After viewing the sensitivity curves, please answer the following questions. **What's the relationship between pruning sparsity and model accuracy? (i.e., does accuracy increase or decrease when sparsity becomes higher?)**

Solution: The relationship between pruning sparsity and model accuracy is inverse. When sparsity becomes higher, the model accuracy decreases.

iv. **Do all the layers have the same sensitivity?**

Solution: No.

v. **Which layer is the most sensitive to the pruning sparsity?**

Solution: The first convolution layer (‘backbone.conv0‘).

vi. (Optional) After completing part 7 in the notebook, please answer the following questions. **Explain why removing 30 percent of channels roughly leads to 50 percent computation reduction.**

Solution: The most layers in the given neural network are convolution layers. For convolution, $\#MACs = c_o \cdot h_o \cdot w_o \cdot k_h \cdot k_w \cdot c_i$. Removing 30 percent of channels leads to $\#MACs' = (1-0.3) \cdot c_o \cdot h_o \cdot w_o \cdot k_h \cdot k_w \cdot (1-0.3) \cdot c_i = 0.49\#MACs$. Therefore, removing 30 percent of channels roughly leads to 1-0.49≈50 percent computation reduction.

vii. (Optional) **Explain why the latency reduction ratio is slightly smaller than computation reduction.**

Solution: The latency mostly depends on both computation time and data movement time. For convolution, removing 30 percent of channels does not reduce the size of activation by half, and thus the data movement time does not reduce by half. For layers including BatchNorm and ReLU, the computation time is linear to the size of input, and thus the latency of these layers does not reduce by half. In addition to these workload, neural network inference also contains some overhead that does not depend on the size of the input, such as function call. Such overhead can not be reduced when removing channels.

viii. (Optional) **What are the advantages and disadvantages of fine-grained pruning and channel pruning? You can discuss from the perspective of compression ratio, accuracy, latency, hardware support (*i.e.*, requiring specialized hardware accelerator), etc.**

Solution: The advantages of fine-grained pruning are that it can achieve higher compression ratio and higer accuracy easily. The disadvantages are that it requires specialized hardware support to gain actual latency reduction.
The advantages of channel pruning are that it can easily achieve lower latency on general-purpose hardware. The disadvantage is the lower compression ratio when maintaining the model accuracy.

ix. (Optional) **If you want to make your model run faster on a smartphone, which pruning method will you use? Why?**

Solution: If the mobile phone hardware supports fine-grained pruning, I will use fine-grained pruning since it can prune the model more aggressively and reduce the model size significantly. Otherwise, I will use channel pruning because it is much easier to gain speedup for channel pruning and it is faster than fine-grained pruning on general-purpose hardware.

(b) Please complete `quantization.ipynb`, then answer the following questions.

i. After completing K-means Quantization, please answer the following questions. **If 4-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

Solution: 16.

ii. **If n-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

Solution: 2^n colors.

iii. After quantization aware training we see that even models that use 4 bit, or even 2 bit precision can still perform well. **Why do you think low precision quantization works at all?**

Solution:

1). We can think of quantization as noise. For example, if we were to truncate the decimal in a floating point value, the small fluctuation in value can be seen as noise. As seen before, deep neural networks are good at handling noisy inputs and can still infer patterns from samples with noise.
2). General structure is still maintained after quantization. For example, if we think see a 1 bit image (black or white), we can still make out the shapes and object in that image.
3). When we train with dropout we are essentially using a large ensemble of smaller models. The final values are not coming only from one thing only, but are an amalgamation of many different smaller models. When we quantize the final values, we can consider the central limit theorem as it acts on the ensemble of values. The means will dominate and the variance is tied to how many bits of precision we use.(Fact check with sahai: Given that the mean dominates, the quantization will have less of an affect).

iv. (Optional) Please read through and complete up to question 4 in the notebook, then answer this question.

Recall that linear quantization can be represented as $r = S(q - Z)$. Linear quantization projects the floating point range $[fp_{min}, fp_{max}]$ to the quantized range $[quantized_{min}, quantized_{max}]$.

That is to say,

$$r_{\max} = S(q_{\max} - Z)$$
$$r_{\min} = S(q_{\min} - Z)$$

Substracting these two equations, we have,

$$S = r_{\max}/q_{\max}$$
$$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$$
$$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$$
$$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$$

Which of these is the correct result of subtracting the two equations?

Solution: $S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$

v. (Optional) Once we determine the scaling factor S, we can directly use the relationship between $r_{\min}$ and $q_{\min}$ to calculate the zero point Z.

$$Z = \text{int}(\text{round}(r_{\min}/S - q_{\min})$$
$$Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$$
$$Z = q_{\min} - r_{\min}/S$$
$$Z = r_{\min}/S - q_{\min}$$

Which of these are the correct zero point?

Solution: $Z = \text{int}(\text{round}(q_{\min} - r_{\min}/S))$

vi. (Optional) After finishing question 9 on the notebook, **please explain why there is no ReLU layer in the linear quantized model.**

Solution: Before deployment, Convolution and BatchNorm are fused. During activation quantization, the scale factor S and zero point Z are determined using statics of ReLU output activations. That is, $r_{\min} = 0$. Therefore, the following layer's input activation quantization will naturally clamp the input by 0, which has the same effect as ReLU.

vii. (Optional) After completing the notebook, **please compare the advantages and disadvantages of k-means-based quantization and linear quantization.** You can discuss from the perspective of accuracy, latency, hardware support, etc.

Solution: K-means-based quantization is more flexible since the quantization centroids can be arbitrary floating point values to minimize the quantization error. Therefore, K-means-based quantization can maintain higher accuracy with lower bit widths. However, k-means-based quantization only reduces the storage and still requires floating-point computation. Moreover, decoding the codebook (i.e., memory access on the lookup table) is required before general matrix multiplication on general-purpose hardware like GPUs.

Linear quantization is more hardware-friendly since all weights and activations are stored in integers, and almost all arithmetic operations are integer-based. Therefore, linear quantization can be directly exploited by modern GPUs and CPUs. However, linear quantization requires that quantization centroids are (uniformly distributed) integers, and thus it is much more difficult to maintain accuracy with lower bit-width linear quantization.

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!

We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework? Write it down here where you'll need to remember it for the self-grade form.**

Contributors:

- Anant Sahai.
- Saagar Sanghavi.
- Vignesh Subramanian.
- Josh Sanz.
- Ana Tudor.
- Mandi Zhao.
- Olivia Watkins.
- Suhong Moon.
- Bryan Wu.
- Romil Bhardwaj.
- Yujun Lin.
- Ji Lin.
- Zhijian Liu.
- Song Han.
- Liam Tan.

